IZ*Net

Nicholas Pun

Contents

1 Introduction 1

2 Face Recognition 1
2.1 Method 1
2.2 Results 1
2.3 Lessons Learnt 1

Appendix A Triplet Mining (with painfully too much tech) 3

References 5

1 Introduction

2 Face Recognition

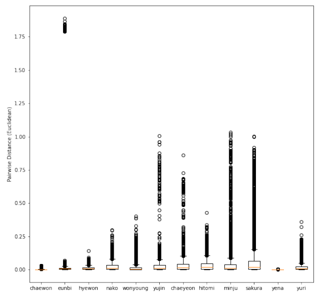
2.1 Method

We explain our methodology for creating a network that will recognize our members:

1. **Creation**: We create NN1 [1, Table 1], NN2 [1, Table 2], Keras InceptionNetV3
2. **Training**: We experiment over two different training techniques: Using the triplet loss function and treating the problem as a binary classification problem [2]
3. **Analysis**: We use a suite of visualizations for analyzing our various trained networks: t-distributed stochastic neighbor embedding, boxplot and a table of medoid distances.

[3] was quite informative for tweaking the hyperparameters of the TSNE algorithm `python`

2.2 Results



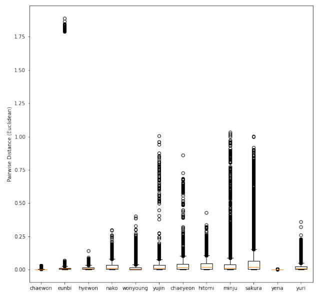


Figure 1: yeet

2.3 Lessons Learnt

1. **blah**

NN1 is not built as specified in [1]. In particular, I had trouble understanding the `fc1` and `fc2` layers, even after reading [4]

A Triplet Mining (with painfully too much tech)

In part of training using triplet loss, we must choose appropriate triplets to feed into our network. There are countless guides and explanations online for how to produce the so-called *semi-hard* and *hard* triplets (over the *easy* ones). But, such is not the focus of this section, instead we are interested in how to choose a uniformly random triplet. Clearly, producing every possible triplet and picking from that set is not very kind to a computer's memory, so the goal is to come up with a scheme that allows to avoid this.

Suppose we have m classes $C_1, \dots, C_m$, not all necessarily the same size. For convenience, define $\mathscr{C}_i = \bigcup_{\substack{k \neq i \\ k \in \{1, \dots m\}}} C_k$. Consider the following selection scheme:

Algorithm 1: Uniform Triplet Mining (Take 1)

```
1 Select $i, j \longleftarrow 1, \dots m$ uniformly at random (just select one and then the other here). $i$ is
    the index of the class that will become the anchor
2 Select an random pair of objects in $C_i$ and a random object from $C_j$
```

Seems simple enough and also not actually uniformly random. This becomes apparent with a small example:

> Say $C_1 = \{\Gamma, \Lambda\}$ and $C_2 = \{\star, \diamondsuit, \square\}$. There are a total of 9 different triplets and more importantly the pair (Γ, Λ) participates in 3 of them. However, using Algorithm 1 above, we have a 100% chance of selecting (Γ, Λ) once we choose C_1 as the anchor, which happen 50% of the time. As such, the triplets involving the pair (Γ, Λ) are represented 50% of the time under this scheme, which is not the $\frac{3}{9} \approx 33.33\%$ we calculated above!

There are two issues with Algorithm 1 above:

1. The selection of j is unnecessary: once a pair has been chosen from C_i, the remaining elements in $\mathscr{C}_i$ are all uniformly represented.

   So, we only need to choose i, select a pair from C_i and then select an object uniformly at random from $\mathscr{C}_i$

2. We cannot assign a uniform distribution to the indices $\{1, \dots, m\}$. Referring back to our example above, note that C_2 produces 3 distinct pairs while C_1 only produces 1. Intuitively, this means we should select C_2 as the anchor more often (in particular, three times as often) so that each pair within the class is equally represented.

   Indeed, let $\mathcal{C} = \binom{C_1}{2} + \dots \binom{C_m}{2}$. Then, set the probability of choosing C_i as the anchor class to be $\frac{\binom{C_i}{2}}{\mathcal{C}}$.

These two modifications give us the following:

Algorithm 2: Uniform Triplet Mining (Take 2)

```
1 Select $C_i$ as the anchor class with probability $\frac{\binom{C_i}{2}}{\mathcal{C}}$
2 Select (uniformly at random) a pair of objects from $C_i$ and (again, uniformly at random) an
    object from $\mathscr{C}_i$
```

It remains to figure out how to actually perform the first step of Algorithm 2. To do this, we use a data structure known as a Segment Tree [1]. We'll leave the details of the data structure for the reader to explore, but in summary:

- A Segment Tree `T` is created from a set of intervals (over $\mathbb{R}$)
- Given a point `v` $\in \mathbb{R}$, the operation `Find(T, v)` returns a list of intervals from `T` containing `v`

We will find that our particular implementation of the Segment Tree is very specialized: we require the intervals to be disjoint and to complete the real line.

With the preparatory work out of the way, we now describe our strategy for choosing the anchor class:

1. Compute the probabilities for each C_i. For convenience, we'll denote these with p_i so we don't have to repeatedly write out the fraction.
2. Give each C_i an interval from $[0, 1)$. This is easy: give C_1 the interval $[0, p_1)$, give C_2 the interval $[p_1, p_1 + p_2)$, and so on.

   Note that $\sum_{i=1}^{m} p_i = 1$, so C_m is given the interval $[p_1 + \ldots + p_{m-1}, 1)$, completing the interval $[0, 1)$
3. Create a Segment Tree `T` from these intervals. Note that we cheat a bit here and give C_1 the interval $(-\infty, p_1)$ and C_m the interval $[p_1 + \ldots + p_{m-1}, \infty)$. This does not affect the probabilities, it is only an consequence of implementation.

Now, pick a random number between `v` $\in [0, 1]$ (say, using the `random` module in Python). The operation `Find(T, v)` will return exactly one interval, and each interval corresponds to exactly one class. As such, `Find(T, v)` selects classes according to the probability distribution we've defined!

This completes Algorithm 2. We end the section with two remarks:

1. As the section title suggests, is painfully too much tech to choose triplets. For the most part, with large enough datasets and close-to-equal distribution of objects between the classes, these steps can be safely ignored. Not using such tooling will skew the data distribution a tad bit, but the effect on the probabilities will be by very *very* small fractional amounts.
2. We employ a similar strategy for choosing pairs for binary classification, which can be found in `PairPickerHelper.py`. The code for triplet picking can be found in `TripletPickerHelper.py`

[1] See `SegmentTree.py`

References

[1] Florian Schroff, Dmitry Kalenichenko, and James Philbin. Facenet: A unified embedding for face recognition and clustering. 2015.

[2] Y. Taigman, M. Yang, M. Ranzato, and L. Wolf. Deepface: Closing the gap to human-level performance in face verification. In *2014 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1701–1708, 2014.

[3] Martin Wattenberg, Fernanda Viégas, and Ian Johnson. How to use t-sne effectively, Oct 2016.

[4] Ian J. Goodfellow, David Warde-Farley, Mehdi Mirza, Aaron Courville, and Yoshua Bengio. Maxout networks. 2013.